A Framework for Specifying and Monitoring User Tasks

Tony Y. Chang
University of Illinois
Urbana, IL 61801
tonychang@acm.org

Neil A. Chilson
University of Illinois
Urbana, IL 61801
chilson@uiuc.edu

Brian P. Bailey
University of Illinois
Urbana, IL 61801
bpbailey@uiuc.edu

ABSTRACT

Knowledge about user task execution can help systems better reason about when to interrupt users. To enable recognition and forecasting of task execution, we develop a novel framework for specifying and monitoring user task sequences. For task specification, our framework provides an XML-based language with tags inspired by regular expressions. For task monitoring, our framework provides an event handler that manages events from any instrumented application and a monitor that observes a user's transitions within and among specified tasks. The monitor supports multiple active tasks and multiple instances of the same task. The use of our framework will enable systems to consider a user's position within a task model when reasoning about when to interrupt.

Keywords

Attention, Interruption, Task Models, Monitoring Tasks

INTRODUCTION

When proactive applications interrupt users at a less opportune moment in their task sequence, it disrupts their task performance [4], error rate [20], decision making [23], and affective state [1] much more than if they had interrupted at a more opportune moment. Prior work has demonstrated that task boundaries represent more opportune moments for interruption that non-boundary moments [1], because users experience less mental workload at those moments [14]. To enable proactive applications to defer information until boundary moments, there needs to be an intelligent mechanism for specifying users' tasks and later monitoring those tasks.

Because existing systems that reason about when to interrupt users rely solely on external and non-task specific interaction cues [12, 13], they could make better decisions with a more precise model of a user's task.

While there has been research on task description languages for generating interfaces and evaluating usability [7, 15, 25], and research on task monitoring for cooperative agents [22, 9], our work provides a unified framework for both specifying and monitoring users tasks.

Our framework consists of a task description language for specifying tasks and patterns of events at multiple levels of detail, an event handler that manages user events from instrumented applications, a graphical tool to lower the specification effort, and a task monitor that follows a user's transitions within and between tasks. Rather than infer task models from user events [19], we provide effective end user tools for specifying tasks with low investment. The task monitor records time spent on each task step, learns transition frequencies among tasks, and stores this information in a model of task execution. The task monitor can notify user-level services when a user begins or finishes a task or subtask and can provide predictions of future task sequences.

In this paper, we describe our task description language and monitoring system. We provide examples illustrating their use and operation. Results are also presented from a user study that shows how our language can be used to specify practical tasks and how our monitoring system can effectively follow a user's progress. Although discussed, learning transition probabilities and providing an appropriate notification API for boundaries is still work in progress. Our framework could be used to instrument the desktop environment as well as correspondence domains for intelligent attention management.

RELATED WORK

We describe how our framework differs from existing scripting languages and frameworks and how its task description language and monitoring system each differ from prior work.

Scripting Languages and Frameworks

A language for scripting user interaction can be used to record macros for automating tasks. A macro thus provides an executable description of a task. Our task description language enables *patterns* of events to be specified and enables other systems to *monitor* user tasks, rather than execute them.

A system-wide scripting framework such as AppleScript [3] enables applications to subscribe to events and access data published by other applications. While these frameworks

[Leave blank the last 2.5 cm (1 inch) of the left-hand column on the first page for the copyright notice.]

enable notification when specific application events occur, our framework enables notification when a user performs and transitions among specified tasks, and can forecast a user's task behavior based on prior observations. Because our framework uses a client/server architecture, it can also interact with applications and services across hetergenous systems.

Task Description Languages

Task description languages have been developed for task analysis [16], to generate interfaces [24], and to predict the usability of interfaces [7, 15, 25]. While these languages can describe different characteristics of user tasks, our task description language enables specification of the hieracrchial decomposition of a task and the pattern of application-level events generated when that task is performed. Our language is also machine parsable and human readable.

Collagen uses task models to link GUI design and agent behavior [22] and the language enables hierarchical decomposition. In contrast, our language uses nesting of XML tags to more naturally express hierarchial decomposition, offers an application-independent language to describe tasks, and uses a syntax and semantics inspired by regular expressions.

ActionStreams [19] is a system that inductively learns hierarchical tasks from the user event stream. The task models can then be used to predict user tasks or inform interface design. While our framework shares similar goals, our approach is to provide an effective set of tools for directly specifying user tasks. This should enable more correct and predictable task specifications, and does not require executing each task multiple times for the underlying system to learn the model.

Intelligent Classroom [9] describes a reactive agent that monitors an instructor's tasks and cooperates by managing media and adjusting camera viewpoints in the classroom. While it shares a similar architecture, our framework provides an effective tool set that enables users to specify their own tasks, targets users interacting with desktop applications, and provides an open architecture to enable user-level services to leverage its task monitoring system.

Task Monitoring

Many systems [22, 19, 9] monitor the user event stream and compare events to a task model. In each case, researchers defined their own system to address this common need. Our work provides a general framework for specifying and monitoring user tasks. Task specific services can be built on top of our framework, saving duplication of effort.

Bayesian networks have been applied to infer a probability distribution over user tasks [2, 10]. The networks typically use specific events or properties of events as evidence variables in the network. This mechanism works well for identifying a task in the midst of sparse or noisy data. In contrast, our monitoring system can monitor multiple ongoing tasks and multiple instances of the same task. Our system also stores how much time a user spends on a task and learns transition probabilities among specified tasks, which Bayesian networks by themselves can not support.

Attentive interfaces seek to monitor a user's changing information needs and offer relevant information [18]. An attentive user interface may benefit from having a more precise understanding of a user's task, which our framework enables, and could leverage our task monitor to better anticipate moments in a user's task where peripheral information could be presented with less disruption.

A challenge in task recognition is how to handle situations where multiple tasks match the same initial sequence of events. In this case, our task monitor creates a candidate set of possible tasks and refines the set as more events are generated. While our approach provides a working solution, more sophisticated, probabilistic approaches such as Dempster-Shafer theory [6] could be used in the future. User preferences for execution sequences and more domain specific information could also help resolve ambiguity in task recognition.

In sum, our work is unique because it utilizes regular expressions and XML nesting to specify user tasks and event patterns using a pithy notation, it can monitor multiple ongoing tasks and forecast task behavior, and it provides an open framework for specifying and monitoring users tasks.

SYSTEM DESIGN GOALS

To guide the development of our system as well as similar systems, we defined design goals for both the task specification language and monitoring system. The term *author* refers to the person writing a task specification, which could be an interface designer, an end user, or other person. Our design goals for the system were to:

- *Enable low-investment specification of user tasks*. The benefit that comes from specifying tasks must outweigh the investment required to specify those tasks. While we have shown the potential benefit of task monitoring for attention management [1], realizing a net benefit requires that a task specification language be easy to use and learn, and be accompanied by effective interface tools that further ease the specification effort.

- *Enable tasks to be specified at multiple levels of detail*. For example, a 'compose email' task could be decomposed into 'open window', 'compose' and 'send mail' subtasks. The 'compose' subtask could then be further decomposed into 'select recipients', 'enter subject', and 'enter body' subtasks, and so forth. For an attention manager, for example, finer task decomposition would enable finer temporal reasoning about when to interrupt [1], but also requires more effort of the author. Striking the appropriate balance between level of detail and specification effort should be left to the author's discretion, not imposed by the system.

- *Enable expressive specification of tasks*. An effective language should enable an author to express variations of task execution in a pithy notation. Although there may be different ways to accomplish a task, an author should not have to *explicitly* describe all those variations. This is analogous to how regular expressions provide a notation that enables a single specification to describe several matching patterns of strings.

- *Enable specification of tasks that cross applications*. User tasks often span applications. An example is that a user receives an email with an attached document, opens the document, edits it, and emails it back to the sender. If performed often, an author may want to specify this sequence as a single task even though it crosses applications.

- *Accurately monitor specified tasks in the midst of unspecified tasks*. A task monitoring system cannot expect all user tasks to be specified due to the enormous space of tasks that a user can perform. Research shows, however, that users often spend about 81% of their time performing a few core tasks in a few applications [8]. Thus, even if a system monitors only a small part of the task space, it is still possible for it to recognize tasks performed most of the time.

- *Support forecasting of a user's task behavior*. By maintaining a historical model of how a user performs and transitions among specified tasks, a system can forecast a user's task behavior. The temporal granularity of the forecasting would be commensurate with the level of detail in the task specifications. For example, for a compose email task specified at a coarse level, a system could forecast that a user will spend about 5 minutes composing an email, or if specified at a finer level of detail, that the user will spend 1 minute selecting recipients, 30 seconds writing the subject, and 3.5 minutes composing the body. Forecasting tasks would be useful, for example, to enable an attention manager to better reason about when to interrupt a user.

FRAMEWORK ARCHITECTURE

As shown in Figure 1, our framework consists of four components; (i) a *task description language* called PETDL that enables an author to express tasks and patterns of events at multiple levels of detail, (ii) an *event handler* that manages user events, (iii) a *graphical tool* called PETDL Maker that can be used to quickly assemble task specifications, and (iv) a *task monitor* that follows a user's progress through a specified task, recording transition frequencies and time spent at each step in a task model.

The framework uses a client/server architecture where the event handler and task monitor can execute on separate machines. This minimizes the performance footprint on client machines, and enables the task monitor to monitor tasks of multiple users.

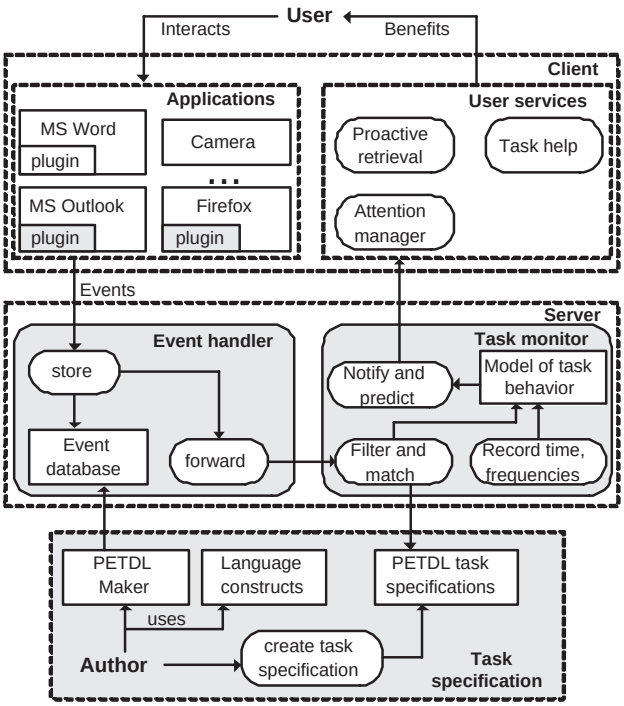

Figure 1: The architecture of our framework.

PETDL Tag	Description
task	Expresses task structure and enables reuse
inOrder	Children tags must occur in specified order
anyOrder	Children tags can occur in any order
optional	Zero or one of specified children may occur
oneOrMore	One or more of specified children may occur
zeroOrMore	Zero or more of specified children may occur
repeatExactly	Children must occur an exact number of times
choice	Exactly one of the children may occur
event	An application-level event

Table 1: PETDL Tags.

In our framework, the term *event* refers to an application-level event, which is a system-level event that has been delivered to and interpreted by an interface control. For example, a system-level event is 'mouse click' while an application-level event is 'save file.' We developed plugins for Outlook and Firefox to demonstrate the feasibility of and test our framework.

Task Description Language

PETDL (**P**attern-based **E**vent and **T**ask **D**escription **L**anguage) is an XML-based language for describing user tasks that draws upon GOMS [7], regular expressions, and schema descriptions. Table 1 shows all the tags available in the language and Figure 2 shows how a calendaring task in Microsoft Outlook can be specified using the tags. Any number of task specifications can be contained in one

```
<task name="Manage Schedule">
  <task name="Schedule Appointment From Email">
    <inOrder>
      <event name="OpenMailItem"/>
      <optional>
        <event name="SwitchFocus"/>
      </optional>
      <anyOrder>
        <task name="AddAppointment">
          <inOrder>
            <event name="OpenApptItem"/>
            <oneOrMore>
              <event name="ChangeApptItemProp"/>
            </oneOrMore>
            <event name="WriteApptItem"/>
            <event name="CloseApptItem"/>
          </inOrder>
        </task>
        <event name="CloseMailItem"/>
      </anyOrder>
    </inOrder>
  </task>
</task>
```

Figure 2: Sample PETDL specification.

PETDL document. PETDL includes tags to describe events, hierarchy, references, and pattern matching as outlined below:

- *Events*. An author uses the `<event>` tag to name an event in a specification. When an application connects to our event handler service, it sends a dictionary of events and their descriptions and stores this information in the event database. Although not shown in Figure 2 for brevity, events may include attributes that specify the name of the application or instance of an application. By including the application name with the event, task specifications can cross application boundaries. The name of the event must exactly match the name of the event forwarded by the application.

- *Hierarchy*. Similar to GOMS [7], PETDL provides a language that an author can use to hierarchically decompose a task into component subtasks (goals) and patterns of events (operators). PETDL provides a single tag `<task>` that can be recursively nested to express subtask hierarchies and to enable subtasks to be reused elsewhere through the use of references. For example, in 2, 'Add Appointment' is a subtask of 'Schedule Appointment From Email.' The use of nesting makes it easier to further decompose existing tasks. In a `<task>` tag, any number of control tags can be used to express patterns of matching user events in the task. The names of the control tags reflect the event patterns they express. The ability to nest task and control tags meets our goal of expressing tasks at multiple levels of detail.

- *References*. A positive consequence of using a nestable tag to express hierarchy is the ability to reuse any subtask in the same or other tasks. This lowers the investment to create and mantain task specifications by reducing duplication of common subtasks and by shortening the PETDL description. PETDL allows a reference to be made to already specified subtasks through the use of a `ref=true` attribute of a task tag. For example, in Figure 3, the 'AddAppointment' task from the beginning of the task is referenced and reused later in the task, and could also be used in separate tasks.

```
<task name="Manage Schedule">
  <task name="AddAppointment">
    <inOrder>
     ...
    </inOrder>
  </task>
  <task name="Schedule Appointment From Email">
    <inOrder>
      ...
      <anyOrder>
        <task name="AddAppointment" ref="true"/>
        <event name="CloseMailItem"/>
      </anyOrder>
    </inOrder>
  </task>
</task>
```

Figure 3: A PETDL specification using a task reference.

- *Pattern Matching*. Listed in Table 1, PETDL provides seven control tags for specifying sequential patterns of user events. Control tags are similar to the control syntax used in regular expressions, e.g., the use of `<zeroOrMore>` in PETDL is equivalent to the use of '*' in regular expressions. Similar to describing matching patterns of strings, an author uses the control tags in a task specification to express matching sequences of user events. Going beyond regular expressions, however, our language also includes an `<anyOrder>` tag. The use of this tag in Figure 2, for example, states that a user may either add the appointment then close the email, or close the email and then add the appointment. The control tags provide a concise and expressive notation for expressing patterns of user events.

Event Handler

The event handler executes in the server process and accepts user events from client applications. For each event, the name of the event, the application that generated it, and the time that it was generated are sent to the handler. The event handler stores this information in an event database to enable a user to inspect and quickly assemble task specifications through the PETDL Maker tool. Once stored, the event handler forwards an event on the the task monitor.

PETDL Maker

PETDL Maker is a graphical tool that enables an author to more easily inspect event dictionaries, inspect event histories, and further ease the task specification effort. It uses a

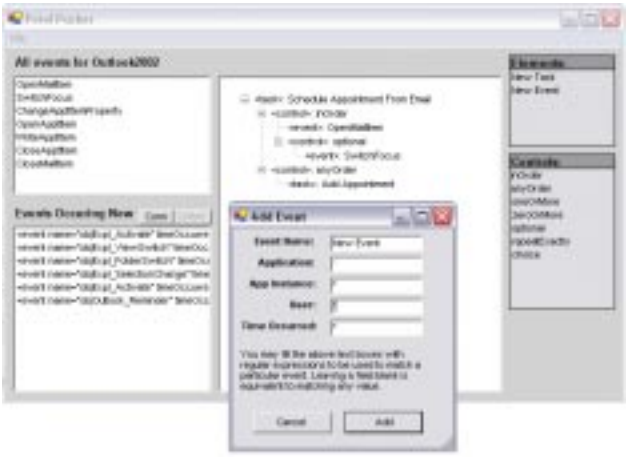

Figure 4: PETDL Maker tool.

drag-and-drop interface that allows users to quickly assemble task specifications by dragging elements from the event dictionary, PETDL tags, and a list of incoming, realtime events from an application. Because the list of events can be large, the tool enables users to filter events using application name and time. To create a specification, an author drags-and-drops elements to the document region, where they are inserted into a tree-structure that enforces and reflects the hierarchical nature of the PETDL language. Because the tool connects to the event database, a user can perform a task, and incorporate the resulting event stream for that task into their PETDL specification. We believe that this macro-style of authoring PETDL documents may dramatically reduce the effort to specify tasks.

Task Monitor

The task monitor can notify user-level services when a user starts or finishes a task or subtask and provide predictions of future task behavior based on historical observations. The task monitor follows a user's progress through PETDL task specifications. It receives events from the event handler and matches the incoming events to the tasks' specifications. The task monitor creates position placeholders for each task that the user is currently in. This allows the user to be in the midst of multiple tasks or the same task multiple times.

When the task monitor receives an event, it is first compared against existing position placeholders to see if it matches the next event in an existing task. If it does not match, it is compared to events that start a new task. If it still does not match, the event is ignored by the task monitor. For example, the event stream in Figure 5 would be matched against the 'Manage Schedule' task of Figure 2 as follows:

1. This first event, `AppActivate`, is ignored by the task monitor because it does not match the first event in the task description, `OpenMailItem`. The task monitor continues to ignore events until it sees `OpenMailItem`.

```
1. <event name="AppActivate"
       timeOccurred="2004.07.04 18:36:09" />
2. <event name="OpenMailItem"
       timeOccurred="2004.07.04 18:36:30" />
3. <event name="ReadMailItem"
       timeOccurred="2004.07.04 18:36:30" />
4. <event name="OpenApptItem"
       timeOccurred="2004.07.04 18:37:53" />
5. <event name="ChangeApptItemProp"
       timeOccurred="2004.07.04 18:38:59" />
6. <event name="ChangeApptItemProp"
       timeOccurred="2004.07.04 18:38:59" />
```

Figure 5: Example event stream with events numbered for reference.

2. The second event, `OpenMailItem`, matches the first event of the "Schedule Appointment From Email" task. Since this is the first event in the task, the task monitor creates a position placeholder to track progress through the task. The next matching event can be `SwitchFocus`, which is optional, or `OpenApptItem` or `CloseMailItem`, which can occur in any order but both must occur.

3. The third event, `ReadMailItem`, does not match the existing position in the task nor does it start a new task. Since it does not match, it is ignored by the task monitor.

4. The fourth event, `OpenApptItem`, gets matched with the position placeholder and now the position placeholder is waiting for `ChangeApptItemProp`.

5. The fifth event, `ChangeApptItemProp`, matches the position placeholder and it is updated and now expecting another `ChangeApptItemProp` or `WriteApptItem`.

6. The last event, `ChangeApptItemProp`, also matches the position placeholder because of the `oneOrMore` control tag. The position placeholder will still match more `ChangeApptItemProp` events or a `WriteApptItem` event.

Because it ignores non-matching events and maintains the position placeholders, the task monitor can monitor tasks in the midst of non-matching events and unspecified tasks. For example, in the 'Manage Schedule' task, the user could save the email at any time - thus generating a non-matching event - without disrupting the task monitor.

Our algorithm compares new events against the events that could possibly occur next. In cases where an event can match multiple subsequent events, an ambiguity arises. We handle this through one of many possible solutions, namely by matching the event to the placeholder that was created first chronologically. If a placeholder has not moved for a long time, then the placeholder is discarded.

As a user transitions among specified tasks, the task monitor builds a model of a user's task execution behavior. The

model is a graph where the nodes represent tasks and edges represent transitions among them. Edges are added as the user transitions between tasks for the first time. Transitions result in the recording of frequency information, i.e. how often the user transitioned to this task versus other possible tasks. From the model, the task monitor can infer the time and the transition to the next task and further task sequences.

IMPLEMENTATION

The task monitor and the event handler were written in Python and together consist of several thousand lines of code. Python's xml.dom library was used to validate PETDL specifications. The task model is a graph built recursively from a PETDL specification. PETDL Maker was written in Visual Basic .NET and consists of about 2,500 lines of code. To help test our framework, plugins were written for MS Outlook and Mozilla Firefox to generate events and send them to the event handler. The plugin for Outlook was written in Visual Basic 6 and monitored events published by the Outlook API. The plugin for Firefox was written in ECMAScript.

EVALUATION

We evaluated how well our language could describe practical, common tasks and whether the task monitor could follow different users performing those tasks. We ran two user studies. The first study was to create task specifications based on observing users performing three tasks. The second study had a different set of users perform the same tasks. We logged events from the second study, compared the event streams to the task specifications, and fed the event streams into the task monitor. This enabled us to measure length of the specifications, how many and which PETDL tags were used, and whether the task monitor could accurately follow different users through specified tasks.

Users and Tasks

Four subjects (one female) participated in the first study, and eight different subjects (four female) participated in the second study. The subjects consisted of undergraduate and graduate students who were experienced users of email, word processing, and web browsing software. In both studies, each user performed the same three tasks.

The first task was a document editing task where a user located an email message in Outlook's inbox, opened the attached document, made corrections, saved the modified document, and sent a reply with the modified document attached. The document was annotated with instructions on how to correct each error. The second task was a web posting task. The user navigated to a website using Mozilla Firefox, found a specific web log entry and posted a comment. The user interacted with the site to ensure that the post was anonymous, to preview the comment, and to make a given change to the comment before making the final post. The third task was a scheduling task where the user opened an email in Microsoft Outlook and scheduled an appointment using Outlook's calendar based on constraints in the email . The user then opened a second email and again scheduled an appointment. Because the constraints were ambiguous, a user may have had to re-schedule the first appointment in order to properly schedule the second appointment

We chose these three tasks because they would provide a sufficient test of our system; they are hierarchical, crossed application boundaries, and are representative of tasks that users often perform. We also designed the tasks so that individual users would likely perform them differently. For example, the constraints in the calendaring tasks sometimes caused users to have to re-schedule the first appointment. This would provide a good test of how well our language and task monitor could handle variance in task execution.

Procedure

In both studies, a user performed practice tasks prior to performing the experimental tasks. After questions were answered, the user performed the experimental tasks. Our plugins intercepted user events and sent them to the event handler for logging. We also recorded a user's screen interaction with Camtasia. Each study lasted no more than 30 minutes.

PETDL Tag	Doc Edit	Web Search	Scheduling	Total
task	4	5	3	9
inOrder	0	0	2	2
anyOrder	0	1	1	2
optional	1	0	1	2
oneOrMore	1	0	1	2
zeroOrMore	0	0	0	0
repeatExactly	0	0	0	0
choice	1	1	0	2
event	10	9	7	24
Total tags	17	16	15	43

Table 2: Tag Frequency

MEASUREMENTS AND RESULTS

From the first study, we reviewed the event logs and screen interaction videos to create PETDL specifications for each of the three tasks. The process was iterative. We created specifications for the first user, adapted them for the second user, and so on, until we had specifications for each task that expressed all users' task sequences. The average length of each task was about 254 seconds and the total time spent constructing a PETDL file was no longer than it took to review the video. After writing the specifications, we counted the frequency of tags used, summarized in Table 2. Overall, the specifications required only a small number of tags to express practical tasks.

Next, we wanted to determine how well the specifications would express different users performing the same tasks. Following a similar process as before, we compared the spec-

ifications and user event streams and classified the outcomes as a match, a user error, or a specification error. A match means that the specification described the event stream. A user error was when the specification could express the event stream, but an error occured due to the user not performing the task as requested. For example, during the web search task, one user posted to the wrong web log. A specification error was when the user performed the task as requested, but the specification did not express the event stream. This was the most serious type of error. Results are shown in Figure 6. Though accuracy was task dependent, matching task execution sequences was good overall.

Specifications were created quickly and were relatively short in length. This demonstrates that creating task specifications is feasible in practice. We also fed the event streams that were properly expressed by the specifications into the task monitor. This was done to determine whether its algorithm could properly match the event streams to the specifications. In each case, the task monitor correctly matched the events to the specifications, validating that its algorithm worked.

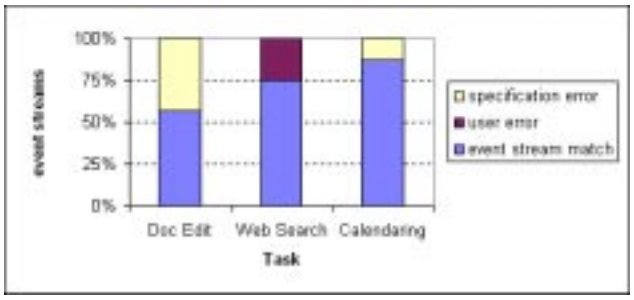

Figure 6: task recognition

DISCUSSION AND FUTURE WORK

We discuss how we met our system design goals. To enable low-investment specification of user tasks, we offer a language that uses a small number of tags that are concise, yet expressive, that results in low-complexity specifications, and is accompanied by a graphical tool to help build specifications. To enable tasks to be specified at multiple levels of detail, we rely on the simple nesting structure inherent in XML. To enable expressive specification of tasks, we provide a small set of tags that can be used to rapidly create variants of common task execution sequences. To enable specification of tasks that cross applications, we allow the names of events to be prefaced by the name or instance of the application. To accurately monitor specified tasks in the midst of unspecified tasks, position placeholders are used to mark where a user is relative to each task specification. This supports multiple active tasks and multiple instances of the same task. To support forecasting of a user's task behavior, we attempt to build sequence probabilities based on how a user transitions through a task model. However, this final design goal is still work in progress.

From our experience using the system, we also learned lessons about how to better specify tasks. First, tasks should not end with control tags that may optionally occur. For example, if `<zerOrMore>` or `<oneOrMore>` are used as the last control tag in a task, the task monitor doesn't know whether or not to keep waiting for more repetitions or to mark the task as complete. Second, task authors can help disambiguate task specifications. For example, some tasks can be described with multiple specifications that match the same event sequences. The same problem can be found in regular expressions like `(the)|(this)` which is equivalent to `th((e)|(is))`. While this normally doesn't matter for regular expressions, for task specifications it causes ambiguity when matching events. To help overcome this ambiguity, it is best to group together the longest sequence of events possible in each part of a specification. Another solution could be to review past events and consider future transition probabilities.

Finally, our language enables an author to specify tasks at multiple levels of detail. We found that specifications at a finer level of detail became much more difficult to express (and thus monitor) because there are many more possible task execution sequences. More experience with creating specifications for practical tasks is necessary to understand an appropriate level of detail.

Our future work seeks to further implement algorithms for forecasting a user's task execution sequences, extend the plugins to capture more events, create a user-level service that defers information or attentional cues until task boundaries, and evaluate how much this service mitigates the disruptive effects of interruptions.

CONCLUSION

Knowledge about user task execution can help systems better reason about when to interrupt users. To enable recognition and forecasting of task execution, we developed a framework for specifying and monitoring user task sequences. For task specification, our framework provides an XML-based language with tags inspired by regular expressions. For task monitoring, our framework provides an event handler that manages events from any instrumented application and a monitor that observes a user's transitions within and between specified tasks. We provided examples illustrating their use and operation. Results were also presented from a user study that shows how our language can be used to specify practical tasks and how our monitoring system can effectively follow a user's progress. Our framework could be used to instrument the desktop environment as well as correspondence domains for intelligent attention management.

REFERENCES

1. Adamczyk, P.D. and B.P. Bailey. If Not Now, When? The Effects of Interruption at Various Moments Within Task Execution. *CHI*, 2004, pp. 271-278.

2. Albrecht, D., I. Zukerman, A. Nicholson, and A. Bud. Towards a Bayesian Model for Keyhole Plan Recognition in Large Domains. *Proc. User Modeling*, 1997, pp. 365-376.

3. AppleScript. http://www.apple.com/applescript.

4. Bailey, B.P., J.A. Konstan, and J.V. Carlis. The Effects of Interruptions on Task Performance, Annoyance, and Anxiety in the User Interface. *Proc. INTERACT*, 2001, pp. 593-701.

5. Beek, P. and R. Cohen. Resolving plan ambiguity for cooperative response generation. *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, August 1991, pp. 938-944.

6. Carberry, S. Techniques for Plan Recognition *User Modeling and User-Adapted Interaction*, 11, 2001, pp. 31-48.

7. Card, S., T. Moran, and A. Newell. *The Psychology of Human-computer Interaction* Lawrence Erlbaum and Associates, 1983.

8. Czerwinski M., E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. *CHI*, 2004, pp. 175-182.

9. Franklin D., J. Budzik, and K. Hammond. Plan-based interfaces: keeping track of user tasks and acting to cooperate. *Proc. IUI*, 2002, pp. 79-86.

10. Horvitz, E., J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users. *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, July 1998.

11. Horvitz, E., A. Jacobs and D. Hovel. Attention-Sensitive Alerting. *Proc. Uncertainty and Artificial Intelligence*, 1999, 305-313.

12. Horvitz, E. and J. Apacible. Learning and Reasoning about Interruption. *Proc. Multimodal Interfaces*, 2003.

13. Hudson, S.E., J. Fogarty, C.G. Atkeson, D. Avrahami, J. Forlizzi, S. Kiesler, J.C. Lee and J. Yang. Predicting Human Interruptibility with Sensors: A Wizard of Oz Feasibility Study. *CHI*, 2003, 257-264.

14. Iqbal, S.T., X.S. Zheng and B.P. Bailey. Task-Evoked Pupillary Response to Mental Workload in Human-Computer Interaction. *CHI*, 2004, pp. 1477-1480.

15. Kieras, D.E., S.D. Wood, K. Abotel, and A. Hornof. GLEAN: A Computer-Based Tool for Rapid GOMS Model Usability Evaluation of User Interface Designs. *UIST*, 1995, pp. 91-100.

16. Kirwan, B. and L.K. Ainsworth. A Guide to Task Analysis *Taylor & Francis, Ltd*, August 1992.

17. Lieberman, H. Exploring the Web with Reconnaissance Agents. *Communications of the ACM*, 44 (8): 69-75, 2001.

18. Maglio, P., R. Barrett, C.S. Campbell, and T. Selker. SUITOR: An Attentive Information System. *Proc. IUI*, 2000, pp. 169-176.

19. Maulsby, D. Inductive Task Modeling for User Interface Customization. *Proc. IUI*, 1997, pp. 233-236.

20. McFarlane, D.C. and K.A. Latorella. The Scope and Importance of Human Interruption in HCI Design. *Human-Computer Interaction*, 17 (1): 1-61, 2002.

21. Miyata, Y. and D.A. Norman. The Control of Multiple Activities. In Norman, D.A. and Draper, S.W. (eds.) *User Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, 1986.

22. Rich, C. and C.L. Sidner. COLLAGEN: A Collaboration Manager for Software Interface Agents. *User Modeling and User-Adapted Interaction, Vol. 8, Issue 3/4*, 1998, pp. 315-350.

23. Speier, C., J.S. Valacich and I. Vessey. The Influence of Task Interruption on Individual Decision Making: An Information Overload Perspective. *Decision Sciences*, 30 (2): 337-360, 1999.

24. Szekely, P., P. Luo, and R. Neches. Beyond Interface Builders: Model-Based Interfaces Tools. *Proceedings of INTERCHI*, April 1993, pp. 383-390.

25. Tauber, M.J. ETAG: Extended Task Action Grammar - A language for the description of the user's task language. *Proceedings INTERACT*, Amsterdam, Elsevier, 1990.

26. Welie, M., G.C. van der Veer, and A. Eliens. An Ontology for Task World Models *Proceedings DSV-IS'98*, 1998, pp. 57-70.